Physically Based Rendering (4th Ed): Chapter 2 Summary

A Programmer's Guide to Monte Carlo Integration

Gemini (For a Developer Audience)

November 29, 2025

Introduction: Why Are We Here?

You have a strong programming background, so let's frame this entire chapter not as "calculus," but as **data sampling**.

In computer graphics, we are trying to calculate the color of a pixel. That color is the result of billions of photons bouncing around a room and hitting a camera sensor. To solve this "physically," we need to sum up all the light arriving at that pixel from every possible direction.

In calculus, "summing up an infinite number of things" is an **Integral**.

$$L = \int f(x)\,dx \tag{1}$$

The problem? This integral is impossible to solve analytically. The scene geometry is too complex; you cannot write a clean formula for "a dragon mesh blocking a point light."

Chapter 2 introduces the solution: **Monte Carlo Integration**.

If you cannot solve the equation, you **simulate** it. Instead of calculating the exact area under a curve, you throw thousands of random points at it and average the result. This chapter explains the math (and code) required to do this correctly.

1 Monte Carlo: The Basics

This section bridges the gap between probability theory and code.

1.1 The Random Variable

In your code, you likely use `Math.random()` (or `drand48()`), which returns a floating-point number between 0 and 1. This is a **Uniform Random Variable**, often denoted as ξ (xi) in the book.

- **Uniform**: Every number between 0 and 1 has an equal chance of appearing.
- **Canonical**: This is the raw material we use to build more complex variables.

1.2 The Expected Value

The "Expected Value" $E[x]$ is just the fancy math term for the **average**. If you roll a die 1,000,000 times, the average roll will be 3.5.

$$E[x] = \sum x_i \cdot p_i \tag{2}$$

In programming terms:

```
double sum = 0;
for(int i = 0; i < N; i++) {
    sum += rollDie();
}
double expectedValue = sum / N;
```

1.3 The Monte Carlo Estimator

This is the core formula of the entire rendering engine. The book presents it as:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{3}$$

Let's translate this into a programmer's loop. We want to find the total brightness (Integral) of a point.

- N: The number of samples (rays you shoot).
- $f(X_i)$: The brightness of the light found by ray i.
- $p(X_i)$: The probability that you chose to shoot ray i in that direction.

The Code Implementation:

```
double totalLight = 0.0;
int N = 100; // Number of samples

for (int i = 0; i < N; i++) {
    // 1. Pick a random direction (X_i)
    Vector3 direction = GetRandomDirection();

    // 2. Determine the probability of picking that direction
    double probability = GetProbabilityOfDirection(direction);

    // 3. Trace ray and get light f(X_i)
    double lightIntensity = TraceRay(direction);

    // 4. Accumulate (The Estimator Formula)
    // We divide by probability to weight the sample correctly
    totalLight += lightIntensity / probability;
}

// 5. Average result
double finalPixelColor = totalLight / N;
```

Why divide by probability $p(X_i)$?

Imagine you are trying to find the average height of people in a room, but you only pick people from the basketball team to measure. Your result will be too high (biased).
To correct this, you must **weight** your samples. If you know you are over-sampling tall people (high probability), you divide their contribution by that high probability to "normalize" the result.

- Rare events (low p) get a huge weight (divide by small number).
- Common events (high p) get a small weight.

This ensures the math works out to the correct "Unbiased" average.

1.4 Variance (The Noise)

In your render, "Variance" looks like static or grain.

- **Low Variance**: Smooth image.
- **High Variance**: Noisy, speckled image.

The error in Monte Carlo integration decreases at a rate of $\sqrt{N}$.

- To cut the noise in half, you need **4x** the samples.
- To cut the noise to 1/10th, you need **100x** the samples.

This is why rendering takes so long. The "Optimization" game in rendering is entirely about finding ways to lower variance without just increasing N.

2 Improving Efficiency

Since we can't just set N to infinity, we need smarter ways to sample.

2.1 Stratified Sampling (Jittering)

The Problem: `Math.random()` is "clumpy." If you pick 100 random pixels, you might get 5 pixels right next to each other and leave a huge gap elsewhere.

The Solution: Stratified Sampling. Divide your domain into a grid. Pick **one** random sample inside **each** grid cell.

Code Analogy: Instead of:

```
float u = RandomFloat(); // 0.0 to 1.0
```

Do this (for 4 samples):

```
// Sample 1: Random in 0.00-0.25
// Sample 2: Random in 0.25-0.50
// Sample 3: Random in 0.50-0.75
// ...
```

This reduces variance because you are guaranteed to "look" at every part of the image/light function.

2.2 Importance Sampling

This is the most critical concept in modern rendering.

Analogy: If you are trying to calculate the average brightness of a night sky, and you shoot rays randomly, 99% of them will hit the black sky (0 light). You waste computation. It is smarter to shoot rays **only at the moon** (where the light is).

Mathematically: Make $p(x)$ (your probability of sampling) match $f(x)$ (the brightness).

- If a spot is bright, sample it often (high p).
- If a spot is dark, sample it rarely (low p).

Recall the estimator: $\frac{f(x)}{p(x)}$. If $f(x)$ is large (bright light) and $p(x)$ is large (we sample there often), the ratio remains constant and stable. If we use uniform sampling (constant p) but $f(x)$ spikes, the result varies wildly (high noise).

2.3 Multiple Importance Sampling (MIS)

What if you have a shiny table (glossy material) reflecting a bright light?

- **Strategy A**: Sample based on the Table (glossy reflection direction).
- **Strategy B**: Sample based on the Light (direction to light source).

Which is better? If the table is a perfect mirror, Strategy A is better. If the light is huge, Strategy B is better.

MIS says: **Do both.** Shoot some rays using Strategy A, some using Strategy B, and use a "heuristic" (weighted average) to combine them.

$$w_1 = \frac{p_1^2}{p_1^2 + p_2^2} \tag{4}$$

This handles "corner cases" automatically. It prevents "fireflies" (bright white noise pixels).

3 Sampling Using the Inversion Method

Okay, we know we *want* to do Importance Sampling. We want to generate random numbers that follow a specific curve (e.g., "more samples near the center"). But `Math.random()` only gives us a flat (uniform) distribution between 0 and 1. How do we remap 0-1 to a complex shape?

This section introduces three key terms:

3.1 1. The PDF (Probability Density Function)

Think of this as a **Histogram**. It tells you "how likely is it to pick a value here?"

- For a standard `Math.random()`, the PDF is flat (1.0 everywhere).
- For a "bell curve" (Gaussian), the PDF is high in the middle and low at the ends.
- **Constraint**: The area under the PDF curve must equal 1. (Total probability = 100%).

3.2 2. The CDF (Cumulative Distribution Function)

Think of this as the **Running Total** (or Prefix Sum) of the PDF. If you walk along the PDF from left to right adding up the values, you get the CDF.

- It always starts at 0.
- It always ends at 1.
- It is strictly increasing.

3.3 3. The Inversion Method (The Algorithm)

This is the "Magic Trick" to convert a uniform random number into a custom distribution.

The Steps:

1. **Integrate** the PDF to get the CDF.
2. **Invert** the CDF (Swap X and Y).
3. Feed your random number (u) into the Inverse CDF.

Example: Sampling a Linear Ramp
We want to pick numbers such that larger numbers are more likely (a ramp shape).

- **PDF**: $p(x) = 2x$ (for x in 0..1).
- **CDF**: Integrate $2x \to x^2$. $P(x) = x^2$.
- **Inversion**: Solve $y = x^2$ for x. Result: $x = \sqrt{y}$.

The Code:

```
float u = Math.random(); // Uniform 0..1
float sample = sqrt(u);  // Distributed 0..1, but clustered near 1
```

If you run this code, 'sample' will be biased toward 1.0, exactly matching the $2x$ ramp. You have successfully "importance sampled" a linear function.

4 Transforming Between Distributions

The previous section handled 1D numbers. But in 3D rendering, we need to sample **Directions** (2D/3D vectors). If we just pick a random X and random Y, we sample a square. How do we sample a **Disk** or a **Sphere**?

4.1 The Jacobian

When you stretch a rubber sheet, the image painted on it gets distorted. If you transform a coordinate system (e.g., from Cartesian x, y to Polar r, θ), the "density" of your samples changes.

If you blindly sample r (0..1) and θ (0..2π), your samples will clump in the center of the circle. This is because the "bullseye" is tiny, but the outer ring is huge. If you treat r linearly, you are throwing as many darts at the tiny bullseye as you are at the huge outer ring.

To fix this, we need the **Jacobian**, which measures how much the area "stretches." For polar coordinates, the area element is $r\,dr\,d\theta$. The extra factor r tells us: "We need more samples as radius increases."

4.2 Sampling a Unit Disk

To sample a circle uniformly (without clumping in the center), we must account for that stretching. Using the Inversion Method on the PDF $p(r) \propto r$:

- **CDF**: Integrate $r \to r^2$.
- **Invert**: $r = \sqrt{u}$.

Correct Code for Uniform Disk Sampling:

```
float u1 = Math.random();
float u2 = Math.random();

float r = sqrt(u1);         // Push samples outward to counteract clumping
float theta = 2 * PI * u2; // Full rotation

float x = r * cos(theta);
float y = r * sin(theta);
```

If you missed the `sqrt`, your render would be too bright in the center and dark on the edges.

Summary for the Developer

Chapter 2 isn't asking you to solve integrals by hand. It is asking you to:

1. **Think in Averages**: A pixel is just the average of many random rays.
2. **Understand Bias**: If you sample incorrectly, your image will be too bright or too dark. You must divide by the probability to correct this.

3. **Master the Inversion Method**: This is your toolbox for converting `rand()` into `Vector3`. (PDF $\rightarrow$ CDF $\rightarrow$ Inverse $\rightarrow$ Code).

By mastering these tools, you can replace complex calculus equations with simple `for` loops that converge to the correct physical reality.